TECHNISCHE
UNIVERSITÄT
DRESDEN

WORKING WITH KNOWLEDGE GRAPHS

Lecture 2: Rules

Markus Krötzsch

Knowledge-Based Systems, TU Dresden

EDBT Summer School 2019

The Limits of SPARQL

Not all interesting queries can be asked in SPARQL.

Limits by general query structure

- SPARQL cannot measure, count, or return paths[1]
- No complex analytical algorithms (e.g. PageRank)

Limits by complexity

- SPARQL query answering is NL-complete in data complexity (i.e., sub-polynomial)
- Problems that are not in NL cannot be solved by any SPARQL query

Limits by language design

- Even some queries in NL cannot be expressed in SPARQL (see next)

[1] Partly for performance reasons: queries such as “longest path” are NP-hard with respect to the size of the database; even tiny graphs can have astronomic numbers of simple paths.

Transitive subproperties

“Located in” is naturally transitive, so it makes sense to query with *:

```
SELECT ?place WHERE { ?place  eg:locatedIn*  eg:EU }
```

Transitive subproperties

"Located in" is naturally transitive, so it makes sense to query with *:

```
SELECT ?place WHERE { ?place  eg:locatedIn*  eg:EU }
```

"Located in" can have sub-properties like "located on terrain feature" (Wikidata P706) or "located on street" (Wikidata P669), so it makes sense to include them in query:

```
SELECT ?place WHERE {
 ?place  ?locatedInProperty  eg:EU .
 ?locatedInProperty  eg:subPropertyOf*  eg:locatedIn .
}
```

Transitive subproperties

"Located in" is naturally transitive, so it makes sense to query with *:

```
SELECT ?place WHERE { ?place  eg:locatedIn*  eg:EU }
```

"Located in" can have sub-properties like "located on terrain feature" (Wikidata P706) or "located on street" (Wikidata P669), so it makes sense to include them in query:

```
SELECT ?place WHERE {
 ?place  ?locatedInProperty  eg:EU .
 ?locatedInProperty  eg:subPropertyOf*  eg:locatedIn .
}
```

However, SPARQL is not able to combine the two!
("Find all places that are directly or indirectly connected to the EU via an arbitrarily long path of sub-properties of 'located in'.")

For other examples of inexpressibility, see course exercises ("Challenge").

Datalog

A rule-based query language

Datalog is a simple logical language that combines pattern matching (conjunctive queries) with recursion (re-using intermediate results).

Example: The following rules find all places within the EU:

$$
\begin{aligned}
&\texttt{locProperty(eg:locatedIn)}\\
&\texttt{locProperty}(X) \leftarrow \texttt{locProperty}(Y) \land \texttt{subPropertyOf}(X, Y)\\
\texttt{locatedIn}(X, Y) &\leftarrow \texttt{edge}(X, P, Y) \land \texttt{locProperty}(P)\\
\texttt{locatedIn}(X, Z) &\leftarrow \texttt{locatedIn}(X, Y) \land \texttt{edge}(Y, P, Z) \land \texttt{locProperty}(P)\\
\texttt{euPlace}(X) &\leftarrow \texttt{locatedIn}(X, \texttt{eg:EU})
\end{aligned}
$$

We can read these rules as logical implications, where X, Y, and P are universally quantified variables.

Naive Evaluation of Datalog queries

A straightforward way of evaluating Datalog is to apply rules until saturation:

- Given a database instance $\mathcal{I}$ and a set of rules Σ
- we compute a set of derived facts Δ.

A variable substitution θ is a match of a conjunction φ over a set of facts Δ if $\varphi\theta \subseteq \Delta$.

We can describe a naive evaluation as follows:

```
function eval(Σ, I)

01   $\Delta = \mathcal{I}$
02   repeat :
03       applyRules($\Sigma, \Delta$)
04   until $\Delta$ does not change anymore
05   return $\Delta$
```

```
function applyRules(Σ, Δ)

01   foreach $(\psi \leftarrow \varphi) \in \Sigma$ :
02       foreach match $\theta$ of $\varphi$ over $\Delta$
03           $\Delta = (\Delta \cup \psi\theta)$
```

Then eval($\Sigma, \mathcal{I}$) computes the least model of the Datalog program Σ over database $\mathcal{I}$.

Better Evaluation of Datalog queries

Naive evaluation re-computes all inferences in each iteration. A better approach is to organise inferences by iteration step to disregard previously considered matches:

- We compute sets of facts Δ^i for each step $i = 0, 1, 2, \ldots$
- Let $\Delta^{[i,j]} = \bigcup_{k=i}^{j} \Delta^k$

Better Evaluation of Datalog queries

Naive evaluation re-computes all inferences in each iteration. A better approach is to organise inferences by iteration step to disregard previously considered matches:

- We compute sets of facts Δ^i for each step $i = 0, 1, 2, \ldots$
- Let $\Delta^{[i,j]} = \bigcup_{k=i}^{j} \Delta^k$

This leads to the so-called semi-naive evaluation:

```
function eval(Σ, I)

01  $i = 0$      $\Delta^0 = \mathcal{I}$
02  **repeat** :
03      applyRules($\Sigma, \Delta, i$)
04  **until** $\Delta^i = \emptyset$
05  **return** $\Delta^{[0,i]}$
```

```
function applyRules(Σ, Δ, i)

01  $\Delta^{i+1} = \emptyset$
02  **foreach** $(\psi \leftarrow \varphi) \in \Sigma$ :
03      **foreach** match $\theta$ of $\varphi$ over $\Delta^{[0,i]}$ with $\varphi\theta \cap \Delta^i \neq \emptyset$ :
04          $\Delta^{i+1} = (\Delta^{i+1} \cup \psi\theta) \setminus \Delta^{[0,i]}$
05  $i = i + 1$
```

The additional check $\varphi\theta \cap \Delta^i \neq \emptyset$ restricts to matches that use a recently derived fact.

- The result is equal to that of the naive evaluation
- Efficient implementations look only for relevant matches in the first place

Datalog in practice

Dedicated Datalog engines as of 2019 (incomplete):

- VLog Fast in-memory rule engine with bindings to various data sources [AAAI 2016, IJCAR 2018]
- RDFox Fast in-memory RDF database with rule support
- Llunatic PostgreSQL-based implementation of a rule engine
- Graal In-memory rule engine with RDBMS bindings
- SociaLite and EmptyHeaded Datalog-based languages and engines for social network analysis
- DeepDive Data analysis platform with support for Datalog-based language “DDlog”
- DLV Answer set programming engine that is usable on Datalog programs (commercial)
- VadaLog Datalog-based in-memory rule engine (commercial, unreleased)
- E Fast theorem prover for first-order logic with equality; can be used on Datalog as well
- . . .

$\rightsquigarrow$ Extremely diverse tools for very different requirements

Datalog in VLog4j

VLog can be used most conveniently via the Java library **VLog4j** [ISWC 2019].

The previous example could be represented as follows in VLog4j rule syntax:

```
@prefix eg: <http://example.org/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@source triple(3): load-rdf("/some/rdf-file.nt") .
subPropertyOf(?X,?Y) :- triple(?X, rdfs:subClassOf, ?Y) .
 locProperty(eg:locatedIn) .
 locProperty(?X) :- locProperty(?Y), subPropertyOf(?X,?Y) .
locatedIn(?X,?Y) :- triple(?X,?P,?Y), locProperty(?P) .
locatedIn(?X,?Z) :- locatedIn(?X,?Y), triple(?Y,?P,?Z), locProperty(?P) .
     euPlace(?X) :- locatedIn(?X,eg:EU) .
```

- Variables are written as in SPARQL
- Constants can be IRIs or data values (as in RDF), or just plain strings
- Data sources can be loaded explicitly (here: from an RDF file)

See `https://github.com/knowsys/vlog4j-example` for an example program using VLog4j.

Datalog queries on Wikidata

VLog4j can be used to execute Datalog queries on Wikidata, either by importing (partial) graphs from RDF, or by fetching data via SPARQL:

```
@prefix wdqs: <https://query.wikidata.org/> .
@source phdAdvisor(2): sparql(wdqs:sparql,"student,professor",
  "?student wdt:P184 ?professor .") .

acadAncestor(?X,?Y) :- phdAdvisor(?X,?Y) .
acadAncestor(?X,?Z) :- acadAncestor(?X,?Y), acadAncestor(?Y,?Z) .
```

- Fetch student–advisor relations (P184) from Wikidata using SPARQL
- Compute their transitive closure to find all academic ancestors

Negation

Problem: Negation and recursion are notoriously hard to combine.

Explanation (sketch):

- Negation allows us to draw conclusions from the absence of a fact,
- but our conclusions could lead us to conclude that this very fact is true.

Negation

Problem: Negation and recursion are notoriously hard to combine.

Explanation (sketch):

- Negation allows us to draw conclusions from the absence of a fact,
- but our conclusions could lead us to conclude that this very fact is true.

Solution (simplest solution of many): Avoid difficulties by ruling out such cyclic dependencies on the predicate level $\rightsquigarrow$ stratified negation

Example: People with academic ancestor Gauss (Q6722) but not Poisson (Q190772):

```
@prefix wdqs: <https://query.wikidata.org/> .
@prefix wd: <http://www.wikidata.org/entity/> .
@source phdAdvisor(2): sparql(wdqs:sparql,"student,professor",
  "?student wdt:P184 ?professor .") .

acadAncestor(?X,?Y) :- phdAdvisor(?X,?Y) .
acadAncestor(?X,?Z) :- acadAncestor(?X,?Y), acadAncestor(?Y,?Z) .
nPG(?X) :- acadAncestor(?X, wd:Q6722), ~acadAncestor(?X, wd:Q190772) .
```

Existential Rules

Motivation

A challenge:

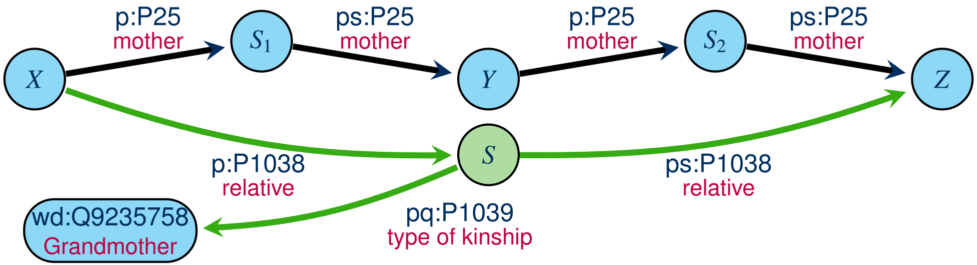
- Datalog can infer new relationships between existing objects,
- but rich graphs like Wikidata represent basic facts by own objects
  $\leadsto$ inferring new Wikidata facts requires adding new objects to the graph!

Motivation

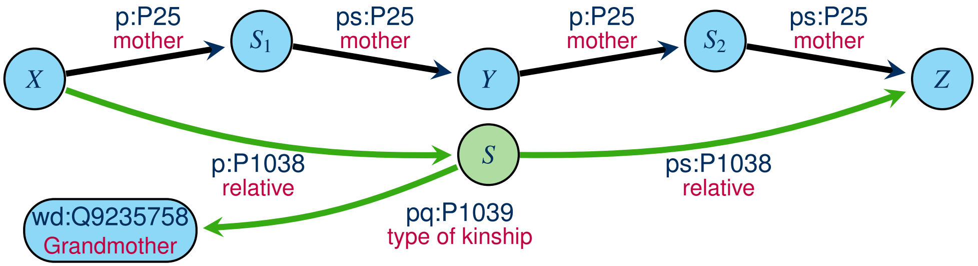
A challenge:

- Datalog can infer new relationships between existing objects,
- but rich graphs like Wikidata represent basic facts by own objects
  $\rightsquigarrow$ inferring new Wikidata facts requires adding new objects to the graph!

Example: Wikidata has no "grandmother" property, but rather represents this relation using property "relative" (P1038) with annotation "type of kinship: grandmother" (P1039: Q9235758).



Existential Rules



Logically, we would like to say something like:

$$\exists S.\ \texttt{p:P1038}(X, S) \land \texttt{ps:P1038}(S, Z) \land \texttt{pq:P1039}(S, \texttt{wd:Q9235758})$$
$$\leftarrow \texttt{p:P25}(X, S_1) \land \texttt{ps:P25}(S_1, Y) \land \texttt{p:P25}(Y, S_2) \land \texttt{ps:P25}(S_2, Z)$$

This is called an existential rule (a.k.a. tuple-generating dependency).

Evaluating rules with existentials

We can adapt the semi-naive evaluation of Datalog to incorporate existential quantifiers.

Idea: create new objects, called fresh nulls, when applying existential rules

```
function chase($\Sigma, \mathcal{I}$)

01  $i = 0$    $\Delta^0 = \mathcal{I}$
02  repeat :
03      applyRules($\Sigma, \Delta, i$)
04  until $\Delta^i = \emptyset$
05  return $\Delta^{[0,i]}$
```

```
function applyRules($\Sigma, \Delta, i$)

01  $\Delta^{i+1} = \emptyset$
02  foreach $(\psi \leftarrow \varphi) \in \Sigma$ :
03      foreach match $\theta$ of $\varphi$ over $\Delta^{[0,i]}$ with $\varphi\theta \cap \Delta^i \neq \emptyset$ :
04          $\theta' = \theta \cup \{\vec{z} \mapsto \vec{n}\}$    // $\vec{z}$ exist. variables in $\psi$; $\vec{n}$ fresh nulls
05          $\Delta^{i+1} = (\Delta^{i+1} \cup \psi\theta') \setminus \Delta^{[0,i]}$
06  $i = i + 1$
```

This algorithm is called the oblivious chase.

Why “chase”?

Applying one rule may lead to new opportunities to apply another rule: we are chasing after a state in which all rules are satisfied

Example: The spouse-relation (P26) is symmetric:

$\exists S.\texttt{p:P26}(Y,S) \wedge \texttt{ps:P26}(S,X) \leftarrow \texttt{p:P26}(X,T) \wedge \texttt{ps:P26}(T,Y)$

Derivations of the oblivious chase:

Why "chase"?

Applying one rule may lead to new opportunities to apply another rule: we are chasing after a state in which all rules are satisfied

Example: The spouse-relation (P26) is symmetric:

$\exists S.\texttt{p:P26}(Y,S) \wedge \texttt{ps:P26}(S,X) \leftarrow \texttt{p:P26}(X,T) \wedge \texttt{ps:P26}(T,Y)$

Derivations of the oblivious chase:

p:P26(taylor, s1234), ps:P26(s1234,burton) (initial facts)

Why "chase"?

Applying one rule may lead to new opportunities to apply another rule: we are chasing after a state in which all rules are satisfied

Example: The spouse-relation (P26) is symmetric:

$\exists S.\texttt{p:P26}(Y,S) \wedge \texttt{ps:P26}(S,X) \leftarrow \texttt{p:P26}(X,T) \wedge \texttt{ps:P26}(T,Y)$

Derivations of the oblivious chase:

p:P26(taylor, s1234), ps:P26(s1234,burton) (initial facts)
p:P26(burton, n_1), ps:P26(n_1,taylor)

Why “chase”?

Applying one rule may lead to new opportunities to apply another rule: we are chasing after a state in which all rules are satisfied

Example: The spouse-relation (P26) is symmetric:

$\exists S.\texttt{p:P26}(Y,S) \wedge \texttt{ps:P26}(S,X) \leftarrow \texttt{p:P26}(X,T) \wedge \texttt{ps:P26}(T,Y)$

Derivations of the oblivious chase:

p:P26(taylor, s1234), ps:P26(s1234,burton) (initial facts)
p:P26(burton, n_1), ps:P26(n_1,taylor)
p:P26(taylor, n_2), ps:P26(n_2,burton)

Why “chase”?

Applying one rule may lead to new opportunities to apply another rule: we are chasing after a state in which all rules are satisfied

Example: The spouse-relation (P26) is symmetric:

$\exists S.\texttt{p:P26}(Y,S) \wedge \texttt{ps:P26}(S,X) \leftarrow \texttt{p:P26}(X,T) \wedge \texttt{ps:P26}(T,Y)$

Derivations of the oblivious chase:

p:P26(taylor, s1234), ps:P26(s1234,burton) (initial facts)
p:P26(burton, n_1), ps:P26(n_1,taylor)
p:P26(taylor, n_2), ps:P26(n_2,burton)
p:P26(burton, n_3), ps:P26(n_3,taylor)

Why “chase”?

Applying one rule may lead to new opportunities to apply another rule: we are chasing after a state in which all rules are satisfied

Example: The spouse-relation (P26) is symmetric:

$\exists S.\mathtt{p{:}P26}(Y,S) \land \mathtt{ps{:}P26}(S,X) \leftarrow \mathtt{p{:}P26}(X,T) \land \mathtt{ps{:}P26}(T,Y)$

Derivations of the oblivious chase:

p:P26(taylor, s1234), ps:P26(s1234,burton)   (initial facts)
p:P26(burton, n_1), ps:P26(n_1,taylor)
p:P26(taylor, n_2), ps:P26(n_2,burton)
p:P26(burton, n_3), ps:P26(n_3,taylor)
...

Why “chase”?

Applying one rule may lead to new opportunities to apply another rule: we are chasing after a state in which all rules are satisfied

Example: The spouse-relation (P26) is symmetric:

$$\exists S.\texttt{p:P26}(Y,S) \land \texttt{ps:P26}(S,X) \leftarrow \texttt{p:P26}(X,T) \land \texttt{ps:P26}(T,Y)$$

Derivations of the oblivious chase:

p:P26(taylor, s1234), ps:P26(s1234,burton) (initial facts)
p:P26(burton, n_1), ps:P26(n_1,taylor)
p:P26(taylor, n_2), ps:P26(n_2,burton)
p:P26(burton, n_3), ps:P26(n_3,taylor)
...

$\leadsto$ the chase may fail to terminate (even if it should)

Everything undecidable

Theorem: All of the following are undecidable.

- Given a set of rules Σ, initial database $\mathcal{I}$, and (variable-free) fact α, decide if α is entailed by Σ over $\mathcal{I}$.
- Given a set of rules Σ and initial database $\mathcal{I}$, decide if the oblivious chase will terminate.
- Given a set of rules Σ, decide if the oblivious chase will terminate over every input database $\mathcal{I}$.

A decidable case

Decidable criteria for detecting chase termination have been studied

Example: Weak acyclicity over-estimates value propagation to exclude cyclic creation of new objects. The oblivious chase then terminates on all databases.

A decidable case

Decidable criteria for detecting chase termination have been studied

Example: Weak acyclicity over-estimates value propagation to exclude cyclic creation of new objects. The oblivious chase then terminates on all databases.

However, weakly acyclic rules are mostly a more concise encoding of Datalog:

	combined complexity	**data complexity**
SPARQL	PSpace-complete	NL-complete
Datalog	ExpTime-complete	P-complete
Weakly acyclic existential rules	2ExpTime-complete	P-complete

The agreement in data complexity reflects a stronger result: anything computable by a weakly acyclic query can also be computed by some Datalog query

These results extend to most other known acyclicity criteria.

Note: answering queries without chase

Chase non-termination does not imply undecidability of query answering!

Alternative query answering approaches exists:

- Bounded treewidth models: compute consequences but apply some complex blocking mechanism to avoid infinite recursion
- Query rewriting: do not compute consequences, but use rules to compute expanded query that can be answered directly
- …

However, the chase is the by far most common algorithm used in rule engines today

A better chase

Idea: We should not introduce new objects if we already have objects that satisfy the entailed facts.

$\leadsto$ restricted chase (a.k.a. standard chase)

```
function chase($\Sigma, \mathcal{I}$)

01  $i = 0$    $\Delta^0 = \mathcal{I}$
02  repeat :
03      applyRules($\Sigma, \Delta, i$)
04  until $\Delta^i = \emptyset$
05  return $\Delta^{[0,i]}$
```

```
function applyRules($\Sigma, \Delta, i$)

01  $\Delta^{i+1} = \emptyset$
02  foreach $(\psi \leftarrow \varphi) \in \Sigma$ :
03      foreach match $\theta$ of $\varphi$ over $\Delta^{[0,i]}$ with $\varphi\theta \cap \Delta^i \neq \emptyset$ :
04          if $\Delta^{[0,i]} \not\models \exists\vec{z}.\psi\theta$ :
05              $\theta' = \theta \cup \{\vec{z} \mapsto \vec{n}\}$  // $\vec{z}$ exist. variables in $\psi$; $\vec{n}$ fresh nulls
06              $\Delta^{i+1} = (\Delta^{i+1} \cup \psi\theta') \setminus \Delta^{[0,i]}$
07  $i = i + 1$
```

The novelty is the check in line 4 of applyRules(), which in practice amounts to query answering over the facts derived so far.

Characteristics of the restricted chase

Some not-so-difficult observations.

Oblivious vs. restricted chase:

- Whenever the oblivious chase terminates, the restricted chase terminates
- The oblivious chase and the restricted chase can have different results
- However, the results are homomorphically equivalent
  $\leadsto$ equivalent for answering positive (negation-free) queries[1]

Non-determinism:

- The exact result of the restricted chase may depend on the order of rule applications
- However, all possible results are homomorphically equivalent and cannot be distinguished by positive queries[1]
- Termination of the restricted chase may depend on the order of rule applications

[1] especially fact-entailment queries

Still everything undecidable

Theorem: All of the following are undecidable.

- Given a set of rules Σ and initial database $\mathcal{I}$, decide if the restricted chase will terminate for some/all rule application strategies.
- Given a set of rules Σ, decide if the restricted chase will terminate over every input database $\mathcal{I}$ for some/all rule application strategies.

Existential rules in VLog4j

VLog4j implements the restricted chase with a Datalog-first rule application strategy: always saturate under Datalog rules before considering rules with existentials

Existential variables are marked by ! in the syntax (now with all prefixes):

```
@prefix wd: <http://www.wikidata.org/entity/> .
@prefix p: <http://www.wikidata.org/prop/> .
@prefix ps: <http://www.wikidata.org/prop/statement/> .
@prefix pq: <http://www.wikidata.org/prop/qualifier/> .
@prefix wdt: <http://www.wikidata.org/prop/direct/> .

p:P1038(?X,!S), ps:P1038(!S,?Z), pq:P1039(!S, wd:Q9235758)
    :- p:P25(?X,?S1), ps:P25(?S1,?Y), p:P25(?Y,?S2), ps:P25(?S2,?Z) .
```

What existentials are good for

We have already seen one well-known example (freshly motivated):
data integration (generating missing structures existentially)

What existentials are good for

We have already seen one well-known example (freshly motivated):
data integration (generating missing structures existentially)

Another possibility has been discovered more recently [ICDT 2019, IJCAI 2019]:
modelling collections (representing sets as explicit objects of the domain)

What existentials are good for

We have already seen one well-known example (freshly motivated):
data integration (generating missing structures existentially)

Another possibility has been discovered more recently [ICDT 2019, IJCAI 2019]:
modelling collections (representing sets as explicit objects of the domain)

Idea:

- A set $\{a, b, c\}$ could be represented by an auxiliary element n with facts

$$\mathtt{in}(a,n) \qquad \mathtt{in}(b,n) \qquad \mathtt{in}(c,n)$$

- Use existential rules to create new sets (with new lists of elements), like so:

$$\exists S.\mathtt{set}(S) \land \mathtt{in}(X,S) \leftarrow \mathtt{makeSingletonSet}(X)$$

What existentials are good for

We have already seen one well-known example (freshly motivated):
data integration (generating missing structures existentially)

Another possibility has been discovered more recently [ICDT 2019, IJCAI 2019]:
modelling collections (representing sets as explicit objects of the domain)

Idea:

- A set $\{a, b, c\}$ could be represented by an auxiliary element n with facts

$$\texttt{in}(a,n) \qquad \texttt{in}(b,n) \qquad \texttt{in}(c,n)$$

- Use existential rules to create new sets (with new lists of elements), like so:

$$\exists S.\texttt{set}(S) \land \texttt{in}(X,S) \leftarrow \texttt{makeSingletonSet}(X)$$

How extend sets by adding elements?

Building bigger sets

A first attempt for adding elements to existing sets:

$$\exists S'.\texttt{set}(S') \land \texttt{plusOneElem}(S,X,S') \leftarrow \texttt{addElement}(X, S)$$
$$\texttt{in}(X,S') \leftarrow \texttt{plusOneElem}(S,X,S')$$
$$\texttt{in}(Y,S') \leftarrow \texttt{plusOneElem}(S,X,S') \land \texttt{in}(Y,S)$$

Building bigger sets

A first attempt for adding elements to existing sets:

$\exists S'.\mathtt{set}(S') \wedge \mathtt{plusOneElem}(S,X,S') \leftarrow \mathtt{addElement}(X, S)$
$\mathtt{in}(X,S') \leftarrow \mathtt{plusOneElem}(S,X,S')$
$\mathtt{in}(Y,S') \leftarrow \mathtt{plusOneElem}(S,X,S') \wedge \mathtt{in}(Y,S)$

Problem: These rules lead to a non-terminating (restricted, Datalog-first) chase:

Example: Consider an input fact `set(emptyset)` and the additional driver rule $\mathtt{addElement}(\mathtt{a},X) \leftarrow \mathtt{set}(X)$, which simply extends every set by element a.

Among others, we get the derivations:

$\mathtt{plusOneElem}(\mathtt{emptyset},\mathtt{a},n_1)$, $\mathtt{plusOneElem}(n_1,\mathtt{a},n_2)$,
$\mathtt{plusOneElem}(n_2,\mathtt{a},n_3)$, $\mathtt{plusOneElem}(n_3,\mathtt{a},n_4)$, ...

This is unavoidable: any correct chase must produce this chain, since positive queries can detect it.

Building bigger sets (another attempt)

Analysis:

- We need facts like `plusOneElem(`S`,`X`,`S'`)` to copy all `in` facts,
- but we need to derive more of them to prevent useless rule applications.

Building bigger sets (another attempt)

Analysis:

- We need facts like $\texttt{plusOneElem}(S,X,S')$ to copy all $\texttt{in}$ facts,
- but we need to derive more of them to prevent useless rule applications.

$$\exists S'.\texttt{set}(S') \wedge \texttt{plusOneElem}(S,X,S') \leftarrow \texttt{addElement}(X,S)$$
$$\texttt{in}(X,S') \leftarrow \texttt{plusOneElem}(S,X,S')$$
$$\texttt{in}(Y,S') \leftarrow \texttt{plusOneElem}(S,X,S') \wedge \texttt{in}(Y,S)$$
$$\texttt{plusOneElem}(S,X,S) \leftarrow \texttt{in}(X,S)$$

Building bigger sets (another attempt)

Analysis:

- We need facts like $\texttt{plusOneElem}(S,X,S')$ to copy all $\texttt{in}$ facts,
- but we need to derive more of them to prevent useless rule applications.

$$
\begin{aligned}
\exists S'.\texttt{set}(S') \wedge \texttt{plusOneElem}(S,X,S') &\leftarrow \texttt{addElement}(X, S) \\
\texttt{in}(X,S') &\leftarrow \texttt{plusOneElem}(S,X,S') \\
\texttt{in}(Y,S') &\leftarrow \texttt{plusOneElem}(S,X,S') \wedge \texttt{in}(Y,S) \\
\texttt{plusOneElem}(S,X,S) &\leftarrow \texttt{in}(X,S)
\end{aligned}
$$

This works:

- facts $\texttt{plusOneElem}(S,X,S)$ prevent the creation of new sets by adding elements
- applying Datalog rules first is essential to create these facts
- termination is guaranteed if the size of our sets is bounded
- using sets in other rules is a two step process:
  (1) infer $\texttt{addElement}(X,S)$ to request creation of a new set
  (2) check for the resulting $\texttt{plusOneElem}(S,X,S')$ to obtain the requested set

How deep is this rabbit hole?

Can we make sets of sets? Sets of sets of sets? . . . and still guarantee termination?

How deep is this rabbit hole?

Can we make sets of sets? Sets of sets of sets? . . . and still guarantee termination?

Yes!

How deep is this rabbit hole?

Can we make sets of sets? Sets of sets of sets? . . . and still guarantee termination?

Yes!

Actually, we get significantly higher expressive power [ICDT 2019]:

	combined complexity	**data complexity**
SPARQL	PSpace-complete	NL-complete
Datalog	ExpTime-complete	P-complete
Weakly acyclic existential rules	2ExpTime-complete	P-complete
Restricted-chase terminating rules	non-elementary	non-elementary

This is the rule language supported by VLog.

Applications

Already computing with sets (of constants) has various applications;

- Ontological reasoning: implement ExpTime-complete description logic reasoning algorithms in (fixed) rule sets [IJCAI 2019]
- Guarded rule reasoning: implement reasoning for guarded existential rules in fixed rule sets [IJCAI 2019]
- Querying for paths: use existential rules to compute paths in knowledge graphs (see exercise)

What we (don't) know

Known knowns: [ICDT 2019]

- The terminating restricted chase is more powerful than the terminating oblivious chase (since non-elementary > PTime)
- The terminating restricted chase is more powerful than the terminating oblivious chase even when considering only PTime queries (surprising!)

Known unknowns:

- Do we gain expressive power by the Datalog-first rule strategy?
- If not: do we gain efficiency?
- What is a good criterion to detect restricted-chase termination?
- Is the terminating restricted chase as powerful as it can get, or is there a more powerful chase algorithm yet?

Unknown unknowns: further open questions await discovery

Summary and conclusions

Rule languages can express graph queries beyond SPARQL

Existential rules add significant capabilities to Datalog:

- data integration (structural expansion of target database)
- set modelling (reasoning with collections of elements)

VLog4j supports existential rule reasoning with stratified negation and SPARQL bindings (and its free and open source! Extensions are welcome!)

The chase algorithm is still only insufficiently understood

Rules offer many worthwhile research topics in theory and practice

(P.S.: We are hiring.)

References and further reading

ICDT 2019	Markus Krötzsch, Maximilian Marx, Sebastian Rudolph: The Power of the Terminating Chase. In Proc. 22nd International Conference on Database Theory (ICDT 2019).
IJCAI 2019	David Carral, Irina Dragoste, Markus Krötzsch, Christian Lewe: Chasing Sets: How to Use Existential Rules for Expressive Reasoning. In Proc. 28th International Joint Conference on Artificial Intelligence (IJCAI'19)
ISWC 2019	David Carral, Irina Dragoste, Larry González, Ceriel Jacobs, Markus Krötzsch, Jacopo Urbani: VLog: A Rule Engine for Knowledge Graphs. In Proc. 18th International Semantic Web Conference (ISWC'19), Springer, to appear.
IJCAR 2018	Jacopo Urbani, Markus Krötzsch, Ceriel Jacobs, Irina Dragoste, David Carral: Efficient Model Construction for Horn Logic with VLog. In Proc. 8th International Joint Conference on Automated Reasoning (IJCAR 2018), Springer.
AAAI 2016	Jacopo Urbani, Ceriel Jacobs, Markus Krötzsch: Column-Oriented Datalog Materialization for Large Knowledge Graphs. In Proc. 30th AAAI Conference on Artificial Intelligence, AAAI Press 2016.
VLog4j	`https://github.com/knowsys/vlog4j`